Predicting Early and Often: Predictive Student Modeling for Block-Based Programming Environments

Andrew Emerson
North Carolina State University
Raleigh, NC 27695
ajemerso@ncsu.edu

Andy Smith
North Carolina State University
Raleigh, NC 27695
pmsmith4@ncsu.edu

Cody Smith
North Carolina State University
Raleigh, NC 27695
crsmit16@ncsu.edu

Fernando J. Rodríguez
University of Florida
Gainesville, FL 32611
fjrodriguez@ufl.edu

Wookhee Min
North Carolina State University
Raleigh, NC 27695
wmin@ncsu.edu

Eric Wiebe
North Carolina State University
Raleigh, NC 27695
wiebe@ncsu.edu

Bradford Mott
North Carolina State University
Raleigh, NC 27695
bwmott@ncsu.edu

Kristy Elizabeth Boyer
University of Florida
Gainesville, FL 32611
keboyer@ufl.edu

James Lester
North Carolina State University
Raleigh, NC 27695
lester@ncsu.edu

ABSTRACT

Recent years have seen a growing interest in block-based programming environments for computer science education. While these environments hold significant potential for novice programmers, they lack the adaptive support necessary to accommodate students exhibiting a wide range of initial capabilities and dispositions toward computing. A promising approach to addressing this problem is introducing adaptive feedback. This work investigates a key capability for adaptive support: training student models that predict student success in block-based programming activities for novice programmers. The predictive student models utilize four categories of features: prior performance, hint usage, activity progress, and interface interaction. In addition to evaluating the accuracy of these models for multiple block-based programming activities, we also investigate how quickly the models converge to accurate prediction, and we evaluate the additive value of each of the four categories of features. Results show that the predictive models are able to predict whether a student will successfully complete an exercise with high accuracy, as well as converge on this prediction early in the sequence of student interactions.

Keywords

Block-Based Programming, Student Performance Prediction, Predictive Student Models

1. INTRODUCTION

A central thrust of computer science education research in recent years has been improving the recruitment and retention of students into computing-related fields of study [2]. Yet, many undergraduate students face challenges in introductory programming courses [36], which have been found to be particularly difficult for novice learners [4]. Block-based programming languages are a promising approach to supporting novices because they reduce the need to focus on syntax, thereby reducing cognitive load and encouraging novices to attempt more complex implementations [40]. In contrast to text-based programming languages, students using block-based programming languages have been shown to spend proportionally more time on productive coding [24]. Further, some aspects of block-based code representations, such as the nesting of blocks and the closer alignment with natural language expression, can help students better understand programming concepts [37]. However, despite their benefits, block-based programming environments have typically provided limited support to students, which places a significant burden on students, as well as their instructors and teaching assistants who have limited availability.

Adaptive learning environments have had success in a broad range of subject matters [19, 20, 32]. They also offer a promising vehicle for addressing the complexity of scaffolding and assessing students' programming activities [3, 6, 8, 15]. A key feature of adaptive learning environments is their ability to leverage student models that assess knowledge and skills from observed learning activities and support learning based on the estimated competency levels in real time.

In this paper, we introduce predictive student models that generate a series of predictions of student success on block-based programming activities. As students complete programming activities, predictive student models can predict student performance, thereby enabling adaptive learning environments to make informed decisions on when to proactively provide support and scaffolding to struggling students. This paper presents predictive student models for block-based programming activities that were trained on over 200 undergraduate students' interactions with a block-based programming environment in an introductory engineering course. The models utilized four categories of student programming behavior: *prior performance*, *hint usage*, *activity progress*, and *interface interaction*. The trained models accurately predict student performance on future programming activities. Analyses of the feature sets reveal prior performance to be the most predictive feature early in the

Andrew Emerson, Andy Smith, Cody Smith, Fernando Rodríguez, Wookhee Min, Eric Wiebe, Bradford Mott, Kristy Boyer and James Lester "Predicting Early and Often: Predictive Student Modeling for Block-Based Programming Environments" In: *Proceedings of The 12th International Conference on Educational Data Mining (EDM 2019)*, Collin F. Lynch, Agathe Merceron, Michel Desmarais, & Roger Nkambou (eds.) 2019, pp. 39 - 48

programming exercises, with other features providing more predictive power as the exercises progress.

This paper is structured as follows. Section 2 discusses related work on analyzing student block-based programs and predicting student coding performance. Section 3 describes PRIME, the block-based programming environment that was used to collect the dataset of students' construction of block-based programs. Section 4 presents the early prediction student modeling framework, as well as an evaluation of the effectiveness of the models, and Section 5 provides a discussion of the results and design implications.

2. RELATED WORK

Hint generation for block-based programming has been the subject of considerable attention. Given the vast solution spaces of programming exercises, data-driven methods of hint generation have been explored extensively. For example, hint generation for Python tutoring [28] as well as comparing the quality of generated hints to expert hints in a block-based programming environment [25] have both shown promise. However, even with high-quality hints, student performance can nevertheless suffer as a result of poor help-seeking behavior, which is prevalent among students who are most in need of assistance [1]. Approaches to hint generation need to address "gaming the system" behaviors, in which students request multiple levels of hints until they receive a "bottom-out" hint [23] providing the answer to the activity. A potential solution is to design a proactive hint generation framework that can monitor students' progress and deliver proactive support when necessary [5, 12]. The work presented in this paper seeks to enable such proactive feedback by creating predictive models capable of accurately detecting struggling students and doing so as early as possible.

Student modeling in programming environments has largely focused on constructing granular models of student domain knowledge. These approaches seek to apply modeling techniques such as Bayesian Knowledge Tracing [7] to programming exercises, mapping exercises to individual knowledge components to identify which concepts students are struggling with [27], and to enable mastery learning [9, 18]. Related work has sought to leverage large datasets and deep neural architectures to analyze student behaviors and identify struggling students as they complete open-ended programming activities [35]. This work builds on these inner-loop student models by incorporating new features such as prior performance and hint usage to predict student performance.

In addition to work on programming environments, examining student behaviors in open-ended environments has shown promise. For example, Sabourin et al. utilized dynamic Bayesian networks to create early prediction models of student learning in a game-based learning environment [31]. Min et al. used deep-learning techniques and multimodal datasets to recognize student goals in an open-ended game-based learning environment [21, 22]. Others have used textbook annotations to create early prediction models of student learning [38], clustering techniques for early prediction of students interacting in an open-ended exploratory simulation environment [11, 16], and fine-grained analysis of game-based learning behaviors to predict student quitting (or dropout) early [17].

3. METHODS

In this work, we investigate college student interactions with a block-based programming environment using features that capture system-level interactions, prior student data, and programming progress. We first describe the learning environment, the coding activities, and interface design, followed by the study with college students and the coding problem-solving dataset collected for training and analysis.

3.1 PRIME Environment

PRIME is an adaptive learning environment designed to support novices in learning computer science concepts through block-based programming. Students can use PRIME both during class time as well as for lab and homework assignments.

3.1.1 Task Progression Design

The curriculum for PRIME was informed by a review of the syllabi for introductory programming courses from the fifty top-rated undergraduate computer science programs in the US [33]. From this review, we identified the set of topics that are typically covered in the first five units of courses, as well as the order in which they are covered: 1) *Input/Output, Variables, and Loops*, 2) *Functions, Parameters, and Return Values*, 3) *Conditional Execution*, 4) *String Manipulation and Basic Data Structures*, and 5) *Search and Sort Algorithms*. The work presented in this paper focuses on Units 1-3 (Table 1).

Table 1. Computer science curricular coverage.

Unit	Topics
1	PRIME environment tutorial, Input/Output, Numeric data types, Expressions (math), Variables, Iteration (definite)
2	Abstraction, Functions (methods), Parameters, Return Values
3	Boolean data types, Conditionals, Iteration (indefinite), Debugging

Each unit of PRIME is typically covered in a week and consists of multiple sequential activities. Units 1 and 2 consist of seven activities each, with Unit 3 consisting of six activities. Within each unit, activities progressively build upon concepts and require students to build more complex programs to solve increasingly challenging problems.

3.1.2 Interface Design

To support the translation of block-based programs into their text-based equivalents (e.g., Python), PRIME uses Google's Blockly framework [13] (Figure 1). The primary user interface provides a *Program* panel, a *Console* panel, a *Feedback* panel, and an *Instructions* panel. The *Program* panel consists of a visual coding widget with the block-based coding workspace and toolbox of available blocks.

The default workspace is augmented with a "Start" block, which serves a purpose analogous to the "main" function or method in other programming languages. The toolbox varies for each task, gradually adding more blocks as students complete tasks and are introduced to new topics. This approach is based on prior work indicating that introducing new blocks only as needed may reduce extraneous cognitive

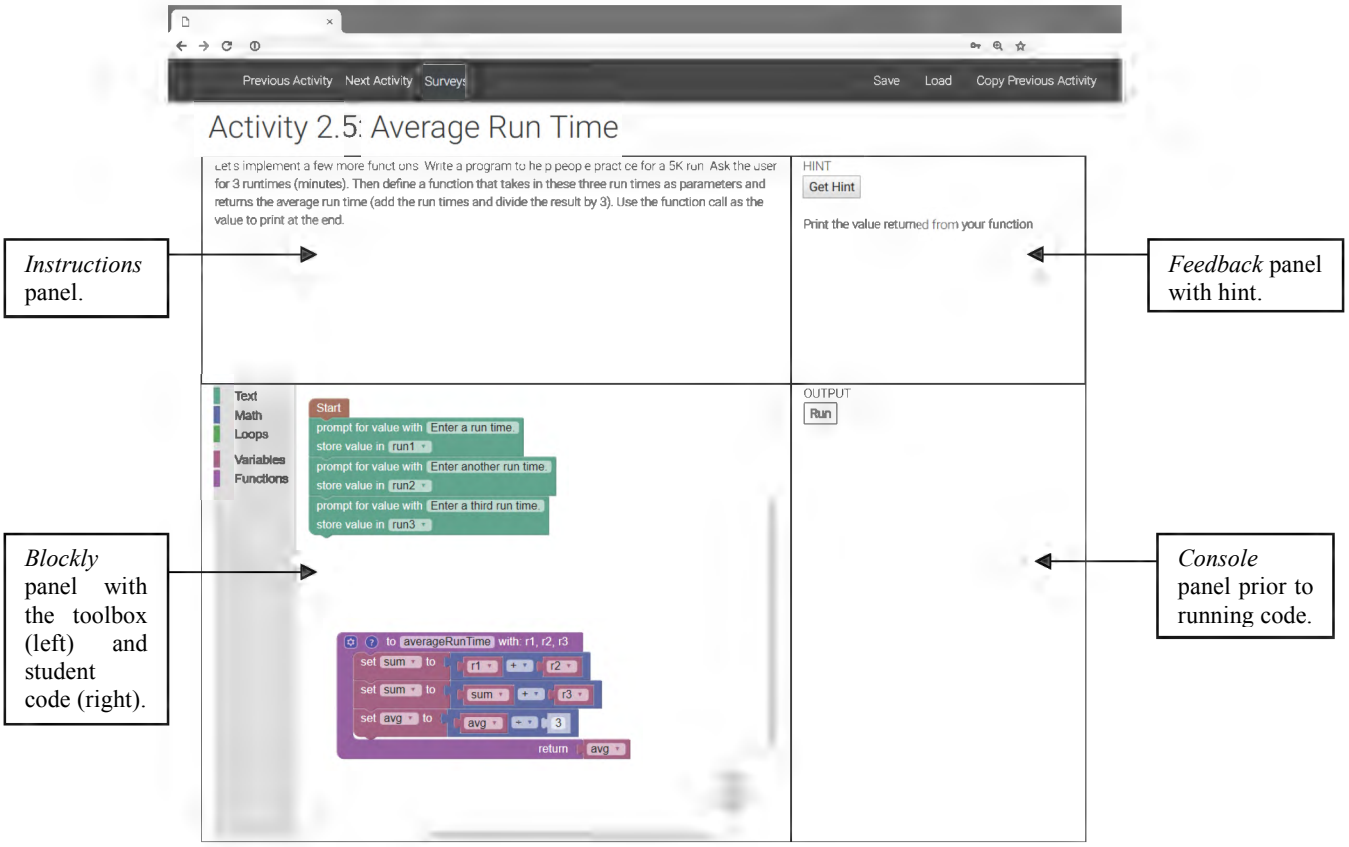

Figure 1. Screenshot of PRIME environment.

load [26] and increase interface usability for novice learners [29].

The *Console* panel contains a "Run" button and shows the output generated from running the program. An input field also appears in this panel if a program prompts the user for input. Finally, the *Instructions* panel contains step-by-step instructions for a given task. This type of instruction format is common for adaptive learning environments [10], though it is rarely found in block-based programming environments. In addition to navigation buttons, this panel also contains positive feedback and links to the next task when a student has successfully completed the current task. Task completeness is checked every time students run their programs and is based primarily on a set of exemplar cases for each activity. Additionally, some activities also check for the presence/absence of certain blocks to ensure an appropriate solution was submitted.

The *Feedback* panel contains the "Get Hint" button, allowing students to request textual hints. Hints are suggestions for minor changes to the program that direct students toward the solution. Hints check various aspects of the student code, including the presence or absence of certain blocks, structural features such as whether code is connected to the "Start" block, and the content of the parameters and fields of certain blocks.

Multiple hints were authored for each activity (M = 5.40, SD = 2.76) based on common errors identified from prior data collections and pilot testing. The maximum number of hints in an activity is 12, and the minimum authored is 2. An example of a hint is, "Instead of numbers, you can put other value blocks (like variables) inside the math operation block." Hints were cast at a sufficiently abstract level that they do not directly provide the solution to an activity but rather nudge students in the right direction. These nudges are designed to assist the student to consider block creations, deletions, or moves that may be advantageous.

Hints are delivered to students in the text panel if they click on the "Get Hint" button in the *Feedback* panel. If no new hints are available, then the button is disabled and cannot be pressed. If there is an available hint different than the one currently displayed, then pressing the button will display that hint. A set of test cases is used to determine which hint is given to the student at a specific point for each activity. The number of test cases passed determines both the specific hint to provide to the student as well as generating the "Next Step" prompt that students receive when completing an intermediate portion of the activity. After requesting a hint, if the student makes changes to satisfy the conditions of the displayed hint, the text of the hint converts to a strikethrough font, visually indicating its status to the student.

3.2 Study Design

Student programming interactions were collected in a study conducted at a large university in the southeastern United States. Participants were students enrolled in two sections of an online introductory course required for all engineering majors. The study sample consisted of 248 students, 222 of which attempted at least one activity.

The average age of the participants was 18, with 31.5% of the group being female. The racial makeup was 75.8% White, 12.9% Asian, 3.2% African American, and 1.2% Hispanic or Latino. The primary major reported was Non-CS Engineering (90.3%), with 6.9% reporting as Computer Science majors and the remaining either Undecided, Math, or Agricultural Science. Of the 222 students, 17.3% reported having prior experience with block-based programming. Of the 222 students who attempted at least one activity, the total number of activities was 2,170 (M = 9.77, SD = 6.24, Median = 8), and the total number of completed activities was 1,492 (M = 6.72, SD = 4.87, Median = 4).

3.3 Dataset

The data used in this study was student interaction data collected from students coding with the PRIME block-based programming environment. The data collected for each student consist of actions the student or system took during the course of an attempted activity. For example, the system logs when students perform actions such as requesting a hint, moving a block, creating a block, and other interactions. When a student performs actions relating to a specific block, the system creates an identification number associated with each block to allow easier tracking of specific blocks that the student creates.

As the goal of this work is to predict successful coding activity completion, the activities used for analysis were selected as those with completion rates between 30-70% (i.e., a 70% threshold for the incompletion/completion rate). Very high completion rates were seen in early activities geared for mastery-oriented introduction to block-based programming. As the activities become more difficult and once the student has been acquainted with the system, the completion rates drop slightly. At the other end of the spectrum, later activities did not have sufficient student attempts for a predictive model to be trained. We therefore focused on the middle activities: 5, 6, 7, 10, 11, 12, 13, 14, 15, 16, 17, and 18. A summary of the completion rates for each activity is shown in Table 2.

Table 2. Completion Rates for each Activity

Activity	*Completed*	*Attempted*	*Success Rate (%)*
5	57	177	32.2
6	61	126	48.4
7	38	108	35.2
10	59	92	64.1
11	46	81	56.8
12	45	71	63.4
13	36	67	53.7
14	32	60	53.3
15	34	78	43.6
16	38	59	64.4
17	39	56	69.6
18	37	55	67.3

3.4 Feature Families

We formulate the task of predictive student modeling for students' coding activities as a binary classification task. We define successful completion of a block-based programming activity as a coding activity that the student completes from start to finish and fulfills each of the activity requirements. Completing an activity will only occur once for each attempt of an activity by a student, and it is important to note that there is no time limit for the completion to occur: a student's interaction with an activity can last from the time they start the activity until the semester ends or until he or she has completed the activity. The input for the predictive models is the number of student-activity attempts where each pair denotes a student attempt on a particular activity. There were a total of 1,966 student-activity attempts in the dataset. This count is calculated as the number of student-activity attempts for which the student had at least 20 system-logged actions within the given activity, where an action is a system-logged interaction where the student clicks within the environment, creates/deletes/moves blocks, or interacts with any of the system components, such as the toolbox or "Save Workspace" button. Thus, the inputs of the predictive models are features derived from each of these student-activity attempts. We define the model input vectors formally with four categories of features: *prior performance*, *hint usage*, *activity progress*, and *interface interaction*. The ***prior performance*** feature is defined as the percentage of activities that a student has completed out of the total activities he or she has attempted up to that point. This feature only considers the activities listed in Table 2. The ***hint usage*** feature is the total number of times a student has requested a hint for the activity (i.e., pressed the "Get Hint" button).

The ***activity progress*** features denote the system analysis of the student's code up to a particular point within the activity. We use two features, *test cases passed* and *checkpoints passed*, to represent the student's progress. These features are calculated by evaluating the student's code with expert-designed test cases. The *test cases passed* feature represents the overall test cases required to complete the activity, and the *checkpoints passed* feature consists of finer-grained test cases within the activity. The *checkpoints passed* feature is used to select which hint to give to the student when a hint is requested. In addition, these intermediate-level test cases are used to drive the "Next Step" prompts that the student sees when making incremental progress required to complete the activity.

The final family of features, ***interface interaction***, consists of 12 features: *enter button presses*, *last save loads*, *previous exercise code loads*, *next instruction clicks*, *previous instruction clicks*, *code runs, workspace saves*, *workspace changes*, *block creations*, *block deletions*, *block moves*, and *user interface clicks in block-display*. These features are logged by the system over the course of a student's attempt at the activity and represent a finer-grained snapshot of the student's problem-solving interactions.

We also use a temporal feature, *time interval*, to introduce a measurement of the student's dynamic progress. We use this to encode sequential interactions and summarize the time interval-based cumulative counts of all other features. The time interval in this work is defined as the 30 second segment of cumulative features up to that point in time (e.g., interval 1 consists of all features within the first 30 seconds, interval 5 consists of the cumulative features up to the first 2 minutes and 30 seconds). There is variation in the maximal interval for

students within the same activity because certain students may take longer to complete (or not complete) the activity. As an example, one student may have spent 2 minutes and 30 seconds on a particular activity, so he or she will have 5 rows of data, each considering the cumulative counts of each described feature. Each row is then indexed by the corresponding time interval, 1 through 5. Before passing the complete input vector into our predictive models, we perform standardization on each feature at the activity level (i.e., subtracting the mean and dividing by the standard deviation).

4. RESULTS

4.1 Activity Completion Predictions

To account for differences in curricular content across activities, we trained a separate model for each activity. We maintained the same set of hyperparameter values for the predictive model for each activity in order to support better generalization to other tasks and to observe patterns spanning all interactions rather than within individual activities. Due to the nature of this predictive task and the fact that the sequential intervals for a particular student-activity attempt are indirectly dependent on one another when comparing the same student, we utilize leave-one-out cross-validation (LOOCV) at the student-level within each activity to validate results found by the predictive models. To report the most accurate results, we averaged the results from the cross-validation process to account for the fact that the student occurring in the test set each iteration does not account for the total distribution of the input data, thereby increasing variance in the results. This validation process was motivated by the occurrence of successive intervals for a given student-activity attempt being related. For example, a student who attempted a specific activity will have cumulative actions in their first interval that will also be counted in the next interval. LOOCV at the student-level thereby prevents data leakage. We adopt logistic regression for interpretability.

Table 3. Classification performance using logistic regression. The results for the majority class baseline (BL), full feature set (Full), and best performing individual family of features (Ind.) are shown.

	Accuracy			*F1*	
Activity	*BL*	*Full*	*Ind.*	*Full*	*Ind.*
5	0.662	0.719	0.666	0.132	0.006
6	0.668	0.751	0.774	0.491	0.469
7	0.708	0.710	0.639	0.281	0.206
10	0.636	0.757	0.815	0.575	0.555
11	0.583	0.762	0.761	0.566	0.563
12	0.747	0.737	0.729	0.680	0.729
13	0.749	0.714	0.664	0.606	0.604
14	0.695	0.704	0.766	0.599	0.611
15	0.607	0.603	0.599	0.417	0.418
16	0.625	0.718	0.694	0.621	0.618
17	0.700	0.865	0.857	0.698	0.683
18	0.757	0.818	0.916	0.701	0.729

We report two metrics: accuracy and F1 score. As a classification problem, it is important to predict both the majority and minority classes at a high rate. In most cases, activity completion is the majority class, but in some cases, the classes are reversed. This occurs primarily when the activities become more difficult, and thus the incompletion rate is greater than the completion rate for that activity. In reporting these metrics, we show both the results from the full set of features (i.e., all four feature categories), and we choose the best single family of features as a comparison. In addition to these, we compare the results against a baseline of the majority class consisting of every time interval where students spent time on an activity. The label for each interval is the end outcome (completion/incompletion) during their interaction with that activity.

Table 3 summarizes the cross-validation results of the full feature sets and the best performing family of features against the majority class baseline. Across all the activities, prior performance was the best performing family of features in 7 out of 12 activities, or 58% of the time. Of the five remaining activities, interface interaction served as the best performing family once, activity progress twice, and hint usage twice.

4.2 Feature Analysis

After determining the best performing model for the entire set of features, it is informative to determine which of the features held the greatest predictive value. After training the model, we evaluated the feature coefficients of the trained regression to determine the relative importance of each of the features. Both the magnitude and sign of the coefficients can be used for interpretation in this case, as we can determine which features were positive or negative predictors in this classification.

Table 4. Logistic regression model coefficients using the full feature set.

Feature	*Mean*	*SD*	*Rank*
Prior Performance	1.269	0.889	1
Time Interval	-1.176	1.493	2
Test Cases Passed	0.828	0.327	3
Block Deletion	0.648	1.486	4
Block Creation	-0.465	0.692	5
Enter Button Presses	0.458	0.528	6
Checkpoints Passed	-0.277	0.251	7
Save Workspace	-0.232	0.780	8
Next Instruction	0.148	0.663	9
Workspace Change	0.118	0.663	10
User Interface Clicks	0.115	0.989	11
Block Moves	-0.094	0.633	12
Hint Button Presses	0.067	0.799	13
Load Last Save	0.063	0.618	14
Load Previous Exercise Code	0.032	0.112	15
Run Code	-0.013	0.896	16
Previous Instruction	0.004	1.022	17

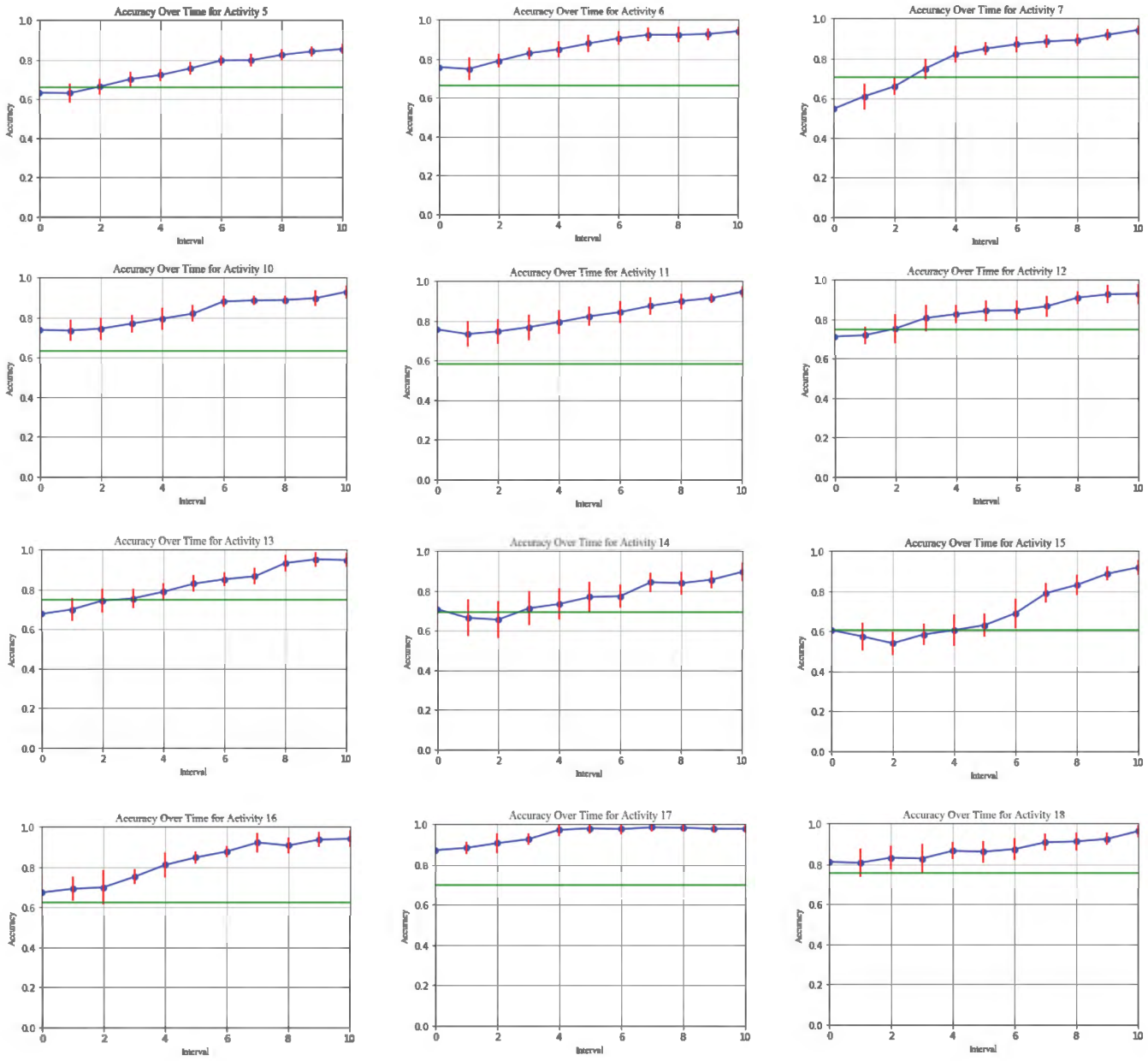

Figure 2. Survival-based analysis of student completion prediction over each interval. The green line represents the baseline for each activity, and the red lines denote the standard deviation of each interval's average accuracy.

In Table 4, the coefficient results for the full feature set are shown. The four strongest predictors in terms of magnitude were *prior performance*, *time interval*, *test cases passed*, and *block deletion*. Many of the features representing students' interactions with the block-based environment (e.g., block creations, deletions) provided a strong boost to predictive performance. In addition, features encoding more productivity-based actions, such as *workspace saves* and *checkpoints passed* also provided an improvement to the model. It is worth noting that *time interval* is a strong negative (coefficient direction) predictor, while prior performance was equally as strong of a positive predictor.

4.3 Early Prediction

As a predictive student model observes more student problem-solving actions over time, we would like for its accuracy to improve. A more rapid convergence toward more accurate predictions would mean that an adaptive learning environment could proactively intervene and provide feedback at an earlier stage if the prediction were that the student would not successfully complete an activity. To evaluate this, we performed a survival-based analysis of the predictive models for each interval of each activity. Specifically, we evaluated the performance of our models at each time interval step, where the accuracy at each successive interval includes the students who have already finished interacting with the

activity as correct predictions. The results for this analysis are shown in Figure 2.

For evaluation, we trained the same logistic regression models on each interval for each activity and recorded the number of errors. The accuracy for each interval is then the number of correct predictions divided by the total number of students who attempted that activity (i.e., the total number of samples for the first interval of that activity). Due to the decreasing size of data for each successive interval, we split the data into a 50% train and 50% test set on each interval for each activity, and we took the average performance over 10 randomly generated splits. We then plotted the accuracy over time, noting the standard deviation as the error bars for each interval. We did not perform LOOCV in this analysis because there is at most one interval per student in each activity, so the train/test split will not have data leakage. In other words, splitting the data in half for a train/test split will not have overlapping students in the test set no matter how the split is made.

As noted above, the desired behavior for these predictive models is that accuracy improves over time as the models observe more student problem-solving interaction data. An additional desirable characteristic is that models surpass the baseline at a relatively fast rate. We note that in 8 of the 12 activities, the accuracy of the first interval is at or above that of the baseline (interval-level class majority). For the remaining activities, and those where the accuracy dips below the baseline, the accuracy surpasses the respective baseline at interval 4, which corresponds to 2 minutes of interaction time with PRIME.

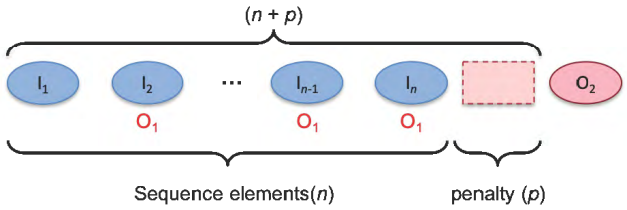

Figure 3. Standardized convergence point metric for sequence prediction analysis.

To quantify the rate at which predictions converged towards an accurate prediction, we also calculated metrics used in the related task of goal recognition for sequence analyses [14, 22]. Specifically, we measured model performance using accuracy rate, convergence rate, convergence point, and standardized convergence point. In this context, *convergence rate* calculates the proportion of sequences where predicted outcome for the final interval is correct. In other words, this metric quantifies how well by the final interval the predictive models can accurately predict whether the student will complete the programming activity. Thus, a higher value for this metric is desirable.

Convergence point refers to the proportion of the sequence of intervals occurring before the predictive model has consistently begun to predict the correct outcome. In other words, this proportion measures a predictive model's ability to converge to an early prediction. This implies that a lower number is more desirable for this metric. The overall convergence point is the average proportion of all sequences of intervals. An issue with using convergence point to measure how early in a sequence predictions converge is that convergence point is only calculated for sequences where the model successfully predicts the last action in a sequence (i.e., a sequence converged to the correct prediction). *Standardized convergence point* (Figure 3) takes this into account by adding a penalty factor for sequences where the last prediction is incorrect. If the prediction of the outcome (O) for a sequence of intervals (I) does not converge, then its value is calculated as $(n + p)/n$, where p is a penalty factor. For this work we set the penalty factor to 1 because of the relatively short length of the sequences investigated. As with convergence point, a lower value for these metrics is desirable.

In Table 5, we report these metrics for our early prediction models using the same train/test split as mentioned previously. Thus, the sequences of intervals for each student-activity attempt in the test set used as the sequences for these metrics. We average the rates for 10 randomly produced train/test splits to validate the results.

Table 5. Averaged rate results for logistic regression (LR) model.

Metric	
Accuracy Rate	63.96%
Convergence Rate	70.62%
Convergence Point	35.84%
Standardized Convergence Point	57.75%
F1 Score	68.87%

5. DISCUSSION

Four families of features contribute to predictive student modeling. *Prior performance*, *hint usage*, *activity progress*, and *interface interactions* all play an important role in accurately predicting student success in programming activities. The predictive models outperform baselines which use majority class prediction of each individual interval. Using these enhanced models, we found that several features stood out as more predictive.

The *prior performance* feature was the strongest positive (coefficient direction) predictor. If students have successfully completed more of their previous exercises, then they are likely to continue doing so. Because this feature accounts for successes on other activities, when students have just begun attempting activities, there will be no data to inform this feature. This is demonstrated in Table 3 when the F1 score is close to 0 for the individual family of features (Ind.) column. However, when no information is known about a student's prior success (i.e., when they are just starting their interaction), the system can use the other features to account for this. A strong negative predictor of student activity completion was the *time interval*. If a student attempts an activity and begins taking more time, the likelihood of their completing that activity may decrease. This could be due to the student not grasping the underlying concept in which the activity is centered.

Two strong positive predictors were *checkpoints passed* and *test cases passed* (activity progress), denoting how many steps a student has completed in the problem. This is a different measure than time interval, as time interval does not capture

the objective progress the student has made. Therefore, if a model is able to detect how much of the code the student has completed relative to the total code needed for the activity, this could boost predictions. The more incremental checkpoints the system designer uses to assess student code, the more likely this feature will be a strong predictor.

Within the interface interaction family of features, *block creations* served as a negative predictor, while *block deletions* served as a positive predictor. These features are fundamental to understanding a student's code. For systems that do not use built-in test cases, these can serve as core predictive features to use for this prediction task. In PRIME, students can freely create, delete, change, and move blocks according to their believed solution to the activity. Actions such as move and create may reveal a more "trial and error" approach, in which the student is attempting new ideas without knowledge of how these blocks interact. Similarly, actions such as deleting a block and changing a block could indicate when a student has tried a block configuration and no longer believes this to be the correct block configuration. In this case, the student is revising his or her answer, and this could point to a block configuration that is closer to an actual solution.

A surprising result is the fact that *hint button presses* was not one of the strongest predictors. The hint request functionality in this environment guides student problem solving at a conceptual level. Hints are designed to nudge students to consider approaches that may spark a correct move or block creation. Thus, if students request many hints, it may be that they are not getting closer to the correct answer. If they keep requesting hints without successfully completing an activity, this could indicate either a lack of effort or lack of understanding, or perhaps both. For effort, analyses would need to be performed to determine if there is a pattern with other interface interaction features that indicate little attempt on the activity. For understanding, analyses would need to be performed to determine if the student is making little progress, such as is the case in wheel-spinning [34]. One reason that hints did not serve as a stronger predictor could be the fact that there were not a consistent number of hints per activity. In addition, these hints are not hierarchical. In other words, the hints do not utilize a "bottom-out" mechanism that becomes finer-grained as the student requests more hints at the same point in their code. This type of hinting system would allow for students who are clearly experiencing an impasse (or lacking effort) to receive more explicit hints, which would likely change the predictive value of the feature.

5.1 Limitations

In this analysis, predictive models were created to determine if a student will complete a block-based programming activity. We explored the possibility of making this prediction as early as possible. While we quantified this through the improvement of accuracy over time and through the use of convergence rate and convergence point, there are no clear standards or baselines for comparing these results. Without a baseline, it is impossible to fully know how this predictive model performs in relation to other models. When determining the type of model to use, we chose logistic regression due to its relative interpretability. However, other models may have higher performance, especially when tuned appropriately. It will be important to investigate other models in future work.

Another limitation is that this analysis did not fully represent the temporal nature of the data. We created a *time interval* feature to account for 30-second intervals, but we do not treat the actions themselves as sequential features. An alternative to treating the features as sequential could be to create finer-grained intervals (varying time lengths) and to use sequential-based machine learning models, such as probabilistic graphical models or recurrent neural networks. Additionally, though the feature families were chosen to generalize well across learning environments, the underlying features are specific to this environment and may not generalize well. Further investigation is needed to understand the effectiveness of this modelling approach for both other programming environments as well as similarly structured environments from other domains.

A final limitation of this work that should be investigated in future work is level of granularity at which analyses of student code is conducted. One way to analyze student code is to perform static tests, such as in *checkpoints passed* and *test cases passed*. Another method would be to create a new representation of the code and analyze this representation, as the automated code analysis approach presented in [39].

6. CONCLUSION

With increasing interest in block-based programming environments for teaching introductory computer science, programming environments that can provide adaptive support hold considerable promise. In order for these environments to evolve beyond providing on-demand hints, there is a need to develop predictive models that can accurately and quickly identify whether a student will succeed or abandon a given activity.

To explore this potential, we created predictive student models for the PRIME block-based programming environment that were informed by four families of features: prior performance, hint usage, activity progress, and interface interaction. Evaluations showed that the models could predict student activity completion more accurately than baselines, and results also demonstrate that by splitting up student-activity attempt data into time intervals, they can make accurate early predictions. A survival-based analysis showed that by 2 minutes of student interaction time, these models consistently outperform baselines, and prior performance, time interval, and test cases passed were the most predictive features.

In future work, it will be important to investigate modeling frameworks that can better leverage sequential features of the data. Second, it will be important to explore more granular assessment rubrics of student programming artifacts, such as those that might be derived from an evidence-centered design approach [30] to drive the predictive models. Third, exploring models that integrate performance prediction with "sibling" models for help-seeking, off-task-behavior, and wheel-spinning is a promising direction for future work. Here, an ensemble of predictive models could be assembled to most effectively support novice student programming. Finally, it will be important to investigate models that operate in tandem with block-based programming and text-based programming and that best support the transition from block-based to text-based programming as students progress to increasingly complex computational problem-solving tasks.

7. ACKNOWLEDGMENTS

This research was supported by the National Science Foundation under Grants DUE-1626235 and DUE-1625908. Any opinions, findings, and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

8. REFERENCES

[1] Aleven, V., Roll, I., McLaren, B.M. and Koedinger, K.R. 2016. Help helps, but only so much: Research on help seeking with intelligent tutoring systems. *International Journal of Artificial Intelligence in Education*. 26, 1, 205–223.

[2] Beaubouef, T. and Mason, J. 2005. Why the high attrition rate for computer science students: Some thoughts and observations. *ACM SIGCSE Bulletin*. 37, 2, 103–106.

[3] Blikstein, P. 2011. Using learning analytics to assess students' behavior in open-ended programming tasks. In *Proceedings of the 1st international conference on learning analytics and knowledge,* 110–116.

[4] Chi, M.T.H. 2005. Commonsense conceptions of emergent processes: Why some misconceptions are robust. *Journal of the Learning Sciences*. 14, 2, 161–199.

[5] Chi, M.T.H., Siler, S. A., Jeong, H., Yamauchi, T. and Hausmann, R.G. 2001. Learning from human tutoring. *Cognitive Science*. 25, 4, 471–533.

[6] Corbett, A., Anderson, J.R. and Patterson, E. 1990. Student modeling and tutoring flexibility in the Lisp intelligent tutoring system. In C. Frasson and G. Gauthier (Eds.). *Intelligent tutoring systems: At the crossroads of artificial intelligence and education*. 83–106. Norwood, NJ: Ablex.

[7] Corbett, A.T. and Anderson, J.R. 1994. Knowledge tracing: Modeling the acquisition of procedural knowledge. *User Modelling and User-Adapted Interaction*. 4, 253–278.

[8] Corbett, A.T. and Anderson, J.R. 2001. Locus of feedback control in computer-based tutoring: Impact on learning rate, achievement, and attitudes. In *Proceedings of the SIGCHI Conference on Human Computer Interaction,* 245–252.

[9] Corbett, A.T. and Anderson, J.R. 1992. Student modeling and mastery learning in a computer-based programming tutor. In *Proceedings of the Second International Conference on Intelligent Tutoring Systems,* 413–420.

[10] Crow, T., Luxton-Reilly, A. and Wuensche, B. 2018. Intelligent tutoring systems for programming education. In *Proceedings of the Twentieth Australasian Computing Education Conference,* 53–62.

[11] Fratamico, L., Conati, C., Kardan, S. and Roll, I. 2017. Applying a framework for student modeling in exploratory learning environments: Comparing data representation granularity to handle environment complexity. *International Journal of Artificial Intelligence in Education*. 27, 2, 320–352.

[12] Gerdes, A., Heeren, B., Jeuring, J. and van Binsbergen, L.T. 2016. Ask-Elle: An adaptable programming tutor for Haskell giving automated feedback. *International Journal of Artificial Intelligence in Education*. 27, 1–36.

[13] Google Blockly - A Visual Programming Editor: 2013.

[14] Ha, E.Y., Rowe, J.P., Mott, B.W., and Lester, J.C. 2012. Goal recognition with Markov logic networks for player-adaptive games. In *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence,* 2113-2119.

[15] Ihantola, P., Edwards, S.H., Petersen, A., Sheard, J., Korhonen, A., Spacco, J., Butler, M., Rivers, K., Szabo, C. and Toll, D. 2016. Educational data mining and learning analytics in programming: Literature review and case studies. In *Proceedings of ACM ITiCSE Conference,* 41–63.

[16] Kardan, S. and Conati, C. 2015. Providing adaptive support in an interactive simulation for learning. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems,* 3671–3680.

[17] Karumbaiah, S., Baker, R.S. and Shute, V. 2018. Predicting quitting in students playing a learning game. In *Proceedings of the International Conference on Educational Data Mining,* 167-176.

[18] Kasurinen, J. and Nikula, U. 2009. Estimating programming knowledge with Bayesian knowledge tracing. *ACM SIGCSE Bulletin*. 41, 3, 313.

[19] Kulik, J.A. and Fletcher, J.D. 2015. Effectiveness of intelligent tutoring systems. *Review of Educational Research*. 37, 1–37.

[20] Ma, W., Adesope, O., Nesbit, J. and Liu, Q. 2014. Intelligent tutoring systems and learning outcomes: A meta-analysis. *Journal of Educational Psychology*. 106, 4, 901–918.

[21] Min, W., Frankosky, M.H., Mott, B.W., Rowe, J.P., Wiebe, E., Boyer, K.E. and Lester, J.C. 2015. DeepStealth: Leveraging deep learning models for stealth assessment in game-based learning environments. In *Proceedings of the Seventeenth International Conference on Artificial Intelligence in Education*, 277–286.

[22] Min, W., Mott, B., Rowe, J., Liu, B. and Lester, J. 2016. Player goal recognition in open-world digital games with long short-term memory networks. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence,* 2590-2596.

[23] Muldner, K., Burleson, W., Van De Sande, B. and Vanlehn, K. 2011. An analysis of students' gaming behaviors in an intelligent tutoring system: Predictors and impacts. *User Modelling and User-Adapted Interaction*. 21, 1–2, 99–135.

[24] Price, T.W. and Barnes, T. 2015. Comparing textual and block interfaces in a novice programming environment. In *Proceedings of the Eleventh International Conference on International Computing Education Research,* 91–99.

[25] Price, T.W., Zhi, R. and Barnes, T. 2017. Hint generation under uncertainty: The effect of hint quality on help-seeking behavior. In *Proceedings of the Eighteenth*

International Conference on Artificial Intelligence in Education, 311–322.

[26] Renkl, A. and Atkinson, R.K. 2003. Structuring the transition from example study to problem solving in cognitive skill acquisition: A cognitive load perspective. *Educational Psychologist*. 38, 1, 15–22.

[27] Rivers, K., Harpstead, E. and Koedinger, K. 2016. Learning curve analysis for programming: Which concepts do students struggle with? In *Proceedings of the Twelfth International Computing Education Research Conference*, 143–151.

[28] Rivers, K. and Koedinger, K.R. 2017. Data-driven hint generation in vast solution spaces: A self-improving Python programming tutor. *International Journal of Artificial Intelligence in Education*. 27, 1, 37–64.

[29] Rodríguez, F.J., Price, K.M., Isaac, J., Boyer, K.E. and Gardner-McCune, C. 2017. How block categories affect learner satisfaction with a block-based programming interface. In *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC* 201–205.

[30] Rupp, A., Levy, R., Dicerbo, K.E., Sweet, S.J., Crawford, A. V., Calico, T., Benson, M., Fay, D., Kunze, K.L., Mislevy, R.J. and Behrens, J. 2012. Putting ECD into practice: The interplay of theory and data in evidence models within a digital learning environment. *Journal of Educational Data Mining*. 4, 1, 49–110.

[31] Sabourin, J., Mott, B. and Lester, J. 2013. Utilizing dynamic Bayes nets to improve early prediction models of self-regulated learning. In *Proceedings of the Twenty-First International Conference on User Modeling, Adaptation and Personalization,* 228–241.

[32] Steenbergen-Hu, S. and Cooper, H. 2013. A meta-analysis of the effectiveness of intelligent tutoring systems on K–12 students' mathematical learning. *Journal of Educational Psychology*. 105, 4, 970–987.

[33] The 50 best computer-science and engineering schools in America: 2015. *http://www.businessinsider.com/best-computer- science-engineering-schools-in-america-2015-7/*.

[34] Wan, H. and Beck, J.E. 2015. Considering the influence of prerequisite performance on wheel spinning. In *Proceedings of the Eighth International Conference on Educational Data Mining*, 129–135.

[35] Wang, L., Sy, A., Liu, L. and Piech, C. 2017. Learning to represent student knowledge on programming exercises using deep learning. In *Proceedings of the Tenth International Conference on Educational Data Mining,* 324–329.

[36] Watson, C. and Li, F.W. 2014. Failure rates in introductory programming revisited. In *Proceedings of the Nineteenth Conference on Innovation & Technology in Computer Science,* 39–44.

[37] Weintrop, D. and Wilensky, U. 2015. Using commutative assessments to compare conceptual understanding in blocks-based and text-based programs. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research,* 101–110.

[38] Winchell, A., Mozer, M., Lan, A., Grimaldi, P. and Pashler, H. 2018. Can textbook annotations serve as an early predictor of student learning? In *Proceedings of the Eleventh International Conference on Educational Data Mining,* 431-437.

[39] Wu, M., Mosse, M., Goodman, N. and Piech, C. 2019. Zero Shot Learning for Code Education : Rubric Sampling with Deep Learning Inference. In *Proceedings of the Thirty-Third International Conference of the Association for Advancement of Artificial Intelligence*.

[40] Xie, B. and Abelson, H. 2016. Skill progression in MIT app inventor. In *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* 213–217.